AN ALGORITHM FOR LINEAR CONSTRAINT SOLVING: ITS INCORPORATION IN A PROLOG META-INTERPRETER FOR CLP

JEAN-LOUIS IMBERT,* JACQUES COHEN,† AND MARIE-DOMINIQUE WEEGER†

▷ The paper presents an incremental and efficient algorithm for testing the satisfiability of systems of linear equalities, inequalities (strict or unrestricted), and disequalities. In addition, it describes the incorporation of that algorithm into a metalevel interpreter capable of processing both tree constraints and the mentioned linear constraints in the domain of rationals. Important characteristics of the described algorithm are (1) detection of fixed variables within the context of Gaussian elimination, including the simplex method, (2) efficient dereferencing by considering subclasses of solved forms, and (3) efficient testing of inconsistencies between equality and disequality subclasses. The metalevel interpreter is written in Prolog. Examples of its usage are provided. Finally, the paper outlines how the approach may be generalized to consider the efficient and incremental testing of constraint satisfiability in various domains. ◁

1. INTRODUCTION

This work combines research in two important areas in constraint language design and implementation: (1) efficiency and incremental testing of satisfiability of linear constraints and (2) metalevel interpretation. It is well known in CLP language design that a desirable characteristic of algorithms for testing satisfiability of constraints is incrementality.[1] This means that given a set of satisfiable constraints C and a new constraint N, the test of satisfiability of $C \cup N$ is carried out as

*Supported by INRIA, France.
†Supported by NSF Grants CCR-87-18989 and CCR-91-15326.
Address correspondence to Professor Jacques Cohen, Mitchtom School of Computer Science, Ford Hall, Brandeis University, Waltham, MA 02254.
Received December 1991; accepted March 1993.
[1] See, for example, [5].

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$6.00

efficiently as possible, using the fact that C is known to be satisfiable. Metalevel interpretation [1] has several useful purposes:

- It provides a concise and rigorous operational semantics of a language.
- It can be used as a guide for efficient implementations.
- It paves the way for the automatic generation of compilers by partial evaluation of interpreters.[2]

The purpose of this paper is (1) to provide a detailed description of an incremental algorithm for processing linear rational constraints involving equalities, inequalities (strict or unrestricted), and disequalities and (2) to present a nucleus of a metalevel interpreter that, in addition to tree constraints, i.e., equalities and disequalities of terms, can also process the constraints in (1).

A novel feature of the algorithm is the minimization of computational costs by restricting the use of general but expensive techniques to the sole cases in which they become unavoidable. For example, the simplex method is only used when strictly necessary and it is applied to the smallest subset of constraints involving nonnegative variables.

This work can be viewed as a substantial part of an ambitious project aiming at providing concise descriptions (using metalevel interpretation) of CLP languages in various domains. The work describing CLP (trees) with equalities and disequalities appears in [10]. The work on linear lists is described in [6]. In addition, we are developing a meta-interpreter for CLP (Booleans). All these subprojects have similar characteristics so that new CLP domains can be incorporated in a modular manner to the existing set of meta-interpreters. The modularized design exemplified in this work can thus be viewed as the second significant contribution of this paper.

It is well known from the simplex method that inequalities can always be transformed into equalities containing variables that are restricted to be nonnegative (slack variables). Strict inequalities, i.e., those containing $>$ and $<$ instead of $\geq$ and $\leq$, can also be transformed into a corresponding set of equalities, supplemented by disequalities specifying that slack variables are restricted to be different from zero.

One of the main contributions of this paper is to distinguish classes of equalities and disequalities that allow an implementor to efficiently and incrementally test the satisfiability of equalities, inequalities, and disequalities. It should be remarked that although current interpreters of constraint languages are based on similar algorithms, the ones proposed here are either more general and/or more efficient than the existing ones [9, 13]. The increase in efficiency is due to the following characteristics of the proposed algorithms: (1) reduction in the number of dereferencing steps and (2) reduction in the number of inconsistency tests. These characteristics are further explained in the sequel.

The incremental testing of satisfiability of a set of linear equalities consists in keeping such equalities in the so-called solved form.[3]

$$variable = constant + \Sigma(coefficient * variable),$$

[2] See [14] and [16].

[3] These solved forms are akin to the canonical forms used by Lassez and McAloon [15].

in which the variables appearing in the r.h.s. differ from those appearing in the l.h.s. Satisfiable values for a l.h.s. variable are obtained by assigning arbitrary values to the r.h.s. variables. Every time a new equality is considered, its variables are "dereferenced," i.e., they are replaced by the corresponding r.h.s. and arithmetic simplifications are made. The resulting dereferenced equality may yield (1) an inconsistency (e.g., $3 \neq 5$), (2) an obvious consistency (e.g., $6 = 6$), or (3) an equality containing known and/or new variables.

In case (1) the test of satisfiability fails, in (2) the test is trivially satisfied, the equality being discarded, and in (3) a variable is chosen, the considered equality is placed in solved form, and added to the list of satisfiable equalities.

One can distinguish the following variants of solved forms:

F_1: A variable v_i is defined as a linear expression that does not contain a *previously defined* variable v_j such that $0 \leq j \leq i$.

F_2: A variable is defined as a linear expression that does not contain *any* of the remaining defined variables.

For example, the definitions (a) $v_1 = 2 - v_2 + v_3$ and (b) $v_2 = v_3$ are in F_1 but not in F_2. Notice that the preceding example contains a fixed hidden variable, i.e., one can infer that $v_1 = 2$. Also note that dereferencing (a) using (b) yields (a') $v_1 = 2$ and (b') $v_2 = v_3$, which is now in F_2 form and the fixed variable is explicit. This example shows that keeping variable definitions in F_2 may be costlier, because it may require a larger number of dereferencing operations. However, the detection of fixed variables is easily performed if the definitions are kept in F_2 form.[4] Also note that, once a fixed variable is detected, substitution of the variable by its value may yield additional fixed variables. This process terminates when no additional fixed variables are detected.

In the following sections, we provide a description of the proposed algorithm and show details of its implementation and the incorporation to a metalevel interpreter. Examples using the interpreter are also presented.

2. PROPOSED ALGORITHM

The algorithm that utilizes two sets of variables in solved form. One contains the definition of arbitrary variables, i.e., those that can be assigned either positive or negative values. The second defines only slack (nonnegative) variables. We refer to slack variables as s-variables and arbitrary variables as x-variables. Note that the r.h.s. defining an x-variable may contain both x- and s-variables whereas s-variables are defined as a positive constant added to a (possibly empty) linear combination of other s-variables.

The strategy used in the proposed algorithm is based on the following considerations:

- Fixed x-variables can always be detected by appropriate dereferencing.
- Fixed x-variables are detected by a variant of the simplex method that uses a lexicographical ordering of variables [18].

[4] The available literature indicates that the CLP($\mathcal{R}$) processor uses an F_2 form [13], whereas Prolog III uses F_1 [7].

- Upon detecting fixed x- or s-variables one can immediately test the consistency of disequations of the form $x \neq c$ or $s \neq c$.

The disequalities submitted to and processed by the satisfiability checker are kept in a number of different classes enabling one to reduce the number of tests necessary to insure consistency or to simplify equalities and disequalities.

The following notation is helpful in explaining the proposed algorithm:

E: Set of equalities of x-variables in solved form.
S: Set of equalities of s-variables in solved form.
D: Set of disequalities.

It will be seen later that the E is kept in a hybrid solved form that is intermediate between F_1 and F_2. The merit of the algorithm lies on subdividing E and D into unique classes that are syntactically distinguishable. The algorithm reduces the number of dereferencing steps done using subclasses in E and in S. Furthermore, they minimize the number of tests necessary by checking the consistency between the E, S, and D classes.

It should be remarked that the simplex method is invoked only when deemed strictly necessary to establish the satisfiability of the system of constraints. This occurs when the dereferenced current equality constraint C contains only s-variables. In that case it is important to use a modified version of the simplex algorithm that, besides testing for the satisfiability of $C \cup S$, is also capable of detecting the presence of fixed s-variables, i.e., those that can only be assigned to a single value. In that case the classes E, D, and S are updated and checked for possible inconsistencies.

The informal description in the previous section can be made more precise by considering the operations using the sets E, S, and D. Recall that E defines x-variables in terms of x- and s-variables. S defines s-variables in terms of s-variables and D expresses disequalities involving s- and x-variables. Let C be the current constraint to be considered. Dereferencing C using E and S yields one of the following cases:

1. An inconsistency: the algorithm fails.
2. A trivial consistency: C can be discarded.
3. A new constraint that can be classified as C_E, C_S, or C_D.

Case 3 is the one that needs to be further considered. An entry in the table

	E	S	D
C_E	σ	σ	τ
C_S	σ	simplex	τ
C_D	σ	σ	σ

denotes the union of the sets specified by a given row and column. Each entry indicates the corresponding algorithm component that needs to be invoked. For example, the union $C_S \cup S$ is tested for satisfiability using the simplex. A σ indicates that the constraint is satisfiable and is incorporated to the set specified by the corresponding subscripts E, S, or D. A τ indicates that, before incorporating a dereferenced C into the appropriate set, it is necessary to test for inconsistencies with the set(s) specifying the complements(s) of the class represented by C. Simplifications may occur before incorporating C to one of those sets.

Notice that, when the simplex is invoked, the detection of a fixed s-variable corresponds to the generation of a new constraint C'_S and further tests have to be carried out to insure the consistency among the S and D constraints. A similar situation occurs when fixed x-variables are detected.

In what follows we refine the foregoing algorithm by considering subclasses E and D constraints. In the following tables, x represents an x-variable, s an s-variable, and c a constant that can be equal to zero. The nonzero coefficients of r.h.s. variables do not appear in the table, but are assumed to exist. Furthermore, the plus sign denotes either additions or subtractions.

E

E_0	$x = c$
E_1	$x = c + x$
E_2	$x = c + x + x + \cdots$ or $x = c + x + s + \cdots$
E_3	$x = c + s$
E_4	$x = c + s + s + \cdots$

D

D_0	$(0 \neq c)$
D_1	$0 \neq c + x$
D_2	$0 \neq c + x + x + \cdots$ or $0 \neq c + x + s + \cdots$
D_3	$0 \neq c + s$
D_4	$0 \neq c + s + s + \cdots$

Remark that:

- Classes E_1 to E_4 parallel classes D_1 to D_4.
- Class D_0 is a fictitious one and represents results of trivially successful or unsuccessful atomic tests.
- Class E_2 differs from E_1 by the existence of at least two variables in the right hand side. This situation parallels those of (D_2, D_1), (E_4, E_3), and (D_4, D_3).

As to the class S, it is sufficient to consider only two subclasses:

S

S_0	$s = c,$	$c \geq 0$
S_1	$s = c + s + \cdots,$	$c \geq 0$

It will be seen later that the class S_0 only exists during the execution of the simplex. As soon as a modified simplex algorithm detects an element of S_0, the corresponding values are substituted in E, S_1, and D.

It should be kept in mind that constraints in one of the E or D subclasses may eventually be transferred to another subclass. Figure 1 indicates the possible transitions among subclasses. The arcs of the graph in Figure 1 are labeled by classes (or union of classes) that are used in making the transitions. For example, the labeled graph indicates that a constraint in E_4 or D_4 only needs to be dereferenced using S, whereas a constraint in E_3 or D_3 only needs to be dereferenced using S_0. Thus by referring to the graph one can accomplish the dereferencing using the minimum number of dereferencing steps.

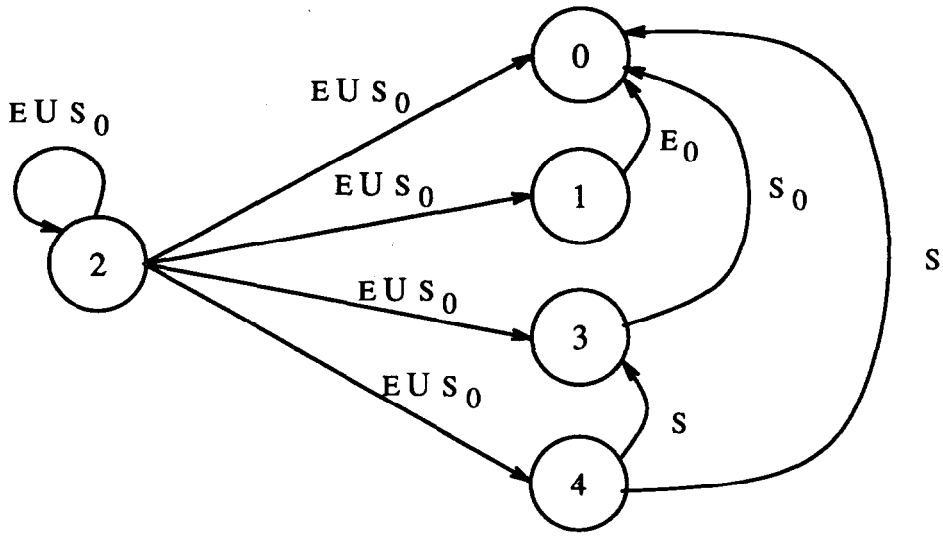

FIGURE 1. Transitions of constraints within E and D Subclasses.

Note that the variables appearing in the r.h.s. of E_2 or E_4 do not appear in the l.h.s. of E_0 to E_4 or S_0, S_1. However, the variable in the r.h.s. of E_1 may appear in one of the definitions in E_1, E_2, E_3, E_4. Also, the variable in the r.h.s. of E_3 may appear in E_4, S_1. This property is insured by dereferencing.[5]

The algorithm in Figure 2 is based on the premise that E, S, and D contain satisfiable constraints. When a new constraint C is considered, the algorithm has two possible outcomes:

1. If $C \cup E \cup S \cup D$ is unsatisfiable, the algorithm fails and E, S, and D remain unchanged.
2. If $C \cup E \cup S \cup D$ is satisfiable, the algorithm succeeds and the new satisfiable sets E', S', and D' are computed.

The algorithm in Figure 2 can be optimized by modifying Steps 2 and 4.1. In Step 2 the dereferencing can be reduced by considering, whenever appropriate, the subclasses within E. The optimized version of Step 2 is presented in Figure 3. Note that if a dereferenced inequality constraint contains only two variables one of which (say x) is not a l.h.s. in E, then no further dereferencing becomes necessary, because the equality can be placed in the form $x = c + x'$, which is satisfiable and belongs to E_1. The placement in E_1 will occur at Step 4.1 (without having to perform Steps 2.2 and 2.3).

The simplex is a well known algorithm whose description appears in linear programming texts, e.g., [4] and [17]. The algorithm can be briefly described as follows. Consider a system $\mathcal{E}$ of m linear equations and n unknowns, with $n > m$. When $\mathcal{E}$ has solutions, they can be obtained by setting $n - m$ of the variables to zero. The remaining m variables can be expressed in a solved form corresponding

[5]Note that the F_1 variant of the solved forms is used for E_1. In contrast E_2, E_3, E_4, and S_1 all use the F_2 variant.

Steps:

1. Consider a new constraint C.
2. Dereference C using E_0 to E_4.
3. Perform arithmetic simplifications (if possible).
4. Classify the dereferenced, simplified C into one of the classes C_E, C_I, and C_D in which C_I represents an inequality. If C_I is strict, it is transformed into the pair of constraints C_E, C_D using a slack variable that is constrained to be positive and different from zero. If C_I is unrestricted, it is transformed into the corresponding C_E using a slack variable constrained to be nonnegative.
   - 4.1. C_E: Dereference the r.h.s. of E_2 and D_2 using C_E. If C_E is an E_0 definition, dereference the r.h.s. of E_1 and D_1 using C_E. Incorporate the dereferenced C_E into the proper class E_0 to E_4. (This dereferencing may transform an E_i definition into another E_j definition. In this case, reenter the algorithm with that constraint.)
   - 4.2. C_D: Incorporate C_D to the corresponding class D_1 to D_4. Note that, prior to incorporating a disequality in D_4, it is necessary to perform a dereferencing using S. If the dereferenced disequality has the form $s \neq c$, then the disequality is satisfiable and belongs to D_3.
   - 4.3. C_S: The simplex algorithm is invoked using $C_S \cup S$. The possible outcomes are:
     - 4.3.1. Unsatisfiability: fail.
     - 4.3.2. Satisfiability without fixed variables: incorporate C_S to S.
     - 4.3.3. Satisfiabiiity with fixed variables: dereference $E \cup S \cup D$ using the constraint $s = c$. The resulting constraints that are not in SF2 are reconsidered. (See Property III in the text.)

     In the case of satisfiability (i.e., 4.3.2 and 4.3.3), E_4 and D_4 have to be dereferenced using S, except if C_S involves a new s-variable.

FIGURE 2. Satisfiability algorithm for linear constraints.

to S_1. These variables are called basic variables and the solution is called a basic solution.

In the actual simplex algorithm one wishes to determine a basic solution minimizing an objective function (a linear function of the variables). For the purposes of a CLP interpreter, the algorithm attempts to minimize the r.h.s. of a new C_S constraint placed in the form $0 = c + s + \cdots + s$ with $c \geq 0$. This minimization is achieved through pivoting, which consists of finding a linear combination of pairs of equations so as to eliminate a given variable. The process of searching for

- 2.1. Dereference C using E_0, E_1.

  Note that:
  - 2.1.1. If a dereferenced equality constraint contains two variables one of which (say, x) is not a l.h.s. in E, then no further dereferencing becomes necessary because the equality can be placed in one of the forms $x = c + x'$ or $x = c + s$, which is satisfiable and belongs to E_1 or E_3.
  - 2.1.2. If a dereferenced disequality constraint has the form $x \neq c$, in which x does not appear in E_0, then the disequality is satisfiable and belongs to D_1.
  - 2.1.3. If a dereferenced inequality constraint has the form $x \geq c$ or $x > c$, the equivalent equality constraint will be of the form $x = c + s$, which is satisfiable and belongs to E_3.

  In all of the foregoing cases, the constraint is placed in the corresponding class (i.e., E_1, E_3, D_1) at Step 4 of the algorithm, without having to perform Steps 2.2 and 2.3.
- 2.2. Simplify if possible.
- 2.3. Classify the dereferenced simplified C into one of the subclasses:
  - —Trivially satisfied or unsatisfied (e.g., $c = c', c \neq c'$), then return.
  - —Otherwise, apply Steps 2.1.2 or 2.1.3 in attempting to proceed directly to Step 4.

FIGURE 3. Optimized Step 2 in Figure 2.

basic solutions by pivoting can result in endless loops. However, precautions can be taken to avoid them [2]. A detailed description of the algorithm appears in [17].

The Step 4.3 in the algorithm of Figure 2 is the modification of the simplex proposed by Van Hentenryck and Graf [18]. The modified algorithm consists of adopting a lexicographical ordering for the variables so that the presence of fixed s-variables can be explicitly detected. An example illustrates this situation. The set of equations

$$\begin{aligned} s'_1 &= 0 + s_1 - s_2, \\ s'_2 &= 0 - s_1 + s_2 \end{aligned} \tag{1}$$

is satisfiable; however, it implies that s'_1 and s'_2 are necessarily equal to zero, i.e., they are fixed variables.

The modified algorithm is based on three properties that are proved in [18]. Before presenting these properties we have to define a solved form (called SF2 by the authors). Consider a linear constraint

$$s' = a_0 + \sum_{i=1}^{n} a_i s_i \tag{2}$$

in which s' and the s_is are restricted to be positive variables, i.e., s-variables. Assume a lexicographical ordering so that

$$s_{i-1} \ll s' \ll s_i.$$

The vector v associated to the constraint (1) is

$$v = [a_0, a_1, \ldots, a_{i-1}, -1, a_i, \ldots, a_n].$$

v is defined as *lexicographically positive* if either (1) all $a_0, a_1, \ldots$ are equal to zero or (2) the first nonzero coefficient a_j, $j \leq i-1$, is positive. A vector v is *lexicographically greater* than a vector u if their difference is positive.

Van Hentenryck and Graf showed that if all the constraints in S can be placed in form (2) so that their v vectors are lexicographically positive, then the constraints are in SF2 form and the following properties hold:

Property I. A system of linear equations in SF2 does not contain hidden fixed s-variables.

Property II. A set of linear equations and disequations is satisfiable iff it can be mapped in SF2.

The first property implies that the solved form in (1) is not in SF2. The determination of fixed variables occurs when (a) C_S has the form

$$0 = \Sigma a_i \, s_i, \tag{2'}$$

where all the a_is are positive or (b) the modified simplex algorithm selects a valid pivot yielding the equation $s = c$. Case (a) indicates that all the s_is must be identical to zero. In that case, the null values are substituted into the remaining solved forms in S_1. As a consequence, other s-variables may be bound to constant values and the process of substitution has to be iterated until no further assignments to constants are produced. The unsatisfiability detected at Step 4.3.1. in Figure 2 occurs when all the coefficients a_i in (2) are positive and a_0 is nonzero.

The detection of hidden fixed s-variables takes place within the simplex: When the classified constraint C_S is considered, it has to be placed in a S_1 form, i.e., a new basic solution has to be determined. This is done by pivoting. In the unmodified simplex there might be a choice for pivots. In the modified simplex the choice of the leaving variable is eliminated by selecting a row having the largest lexicographical vector. Van Hentenryck and Graf call this the *lexicographical rule* and have proved the following property.

Property III. The lexicographic rule preserves SF2 through pivoting.

Property III insures that there are no fixed variables except those occurring in S_0. The algorithm in Figure 2 is based on the following property, which is proved in [11]:

Property IV. There are no hidden fixed x-variables in E except those occurring in E_0.

This property guarantees that any disequation in D_1 (or D_3) will never have an inconsistent counterpart in E_0 (or S_0), i.e., if $0 \neq x - c$ is in D_1, it follows that $x = c$ cannot be in E_0. Similarly, if $0 \neq s - c$ is in D_3, it follows that $s = c$ cannot be in S_0. As a consequence, inconsistencies in D_1 (or D_3) are detected by dereferencing using only E_0 (or S_0), yielding an obvious contradiction such as $s \neq c$.

The following optimization may take place at Step 4.3.3: If prior to entering the simplex a new s-variable is created, then it is not necessary to dereference D_4 unless the new s-variable is found to be fixed, in which case dereferencing of the entire solved form system becomes necessary [11].

Table 1 provides a detailed example of the execution of the proposed algorithm. In that example a selected sequence of linear constraints is used to illustrate the contents of the various classes of solved forms, because the constraints are incrementally considered. The reader may want to verify that the lexicographical rule is indeed used when processing the last constraint in Table 1, thus enabling the detection of the hidden fixed s-variables. Furthermore, that constraint triggers changes in all the subclasses.

3. IMPLEMENTATION

The algorithm described in the previous section is implemented using Prolog. The inherent backtracking capabilities of that language allow one to incorporate the algorithm within the context of a metalevel interpreter.

In this section we consider two aspects of the implementation: (1) the data structures and (2) the major procedures necessary to embody the algorithmic steps presented in Figures 2 and 3.

The data structure is specified by the BNF syntax in Figure 4. The typical examples presented in Figure 5 illustrate the defined data structures. The following remarks are in order:

- Each input variable is given an internal integer code (< *variable-id* >). A list of variables (< *var-list* >) is kept in the form of pairs: the alphanumeric representing the variable and its internal integer code.
- Because new s-variables are needed to process inequalities, the program has to keep track of the last generated s-variable (< *rank-of-last-s-var* >).

TABLE 1. Example

Current Constraint	Class	Solved Form
$2+x_1+2*x_2-x_3-x_5=0$	E_2	$x_1=-2-2*x_2+x_3+x_5$
$2-x_1+x_2\neq 0$	D_2	$4+3*x_2-x_3-x_5\neq 0$
$x_1+2*x_2-x_3\neq 0$	D_1	$-2+x_5\neq 0$
$-1+x_1\neq 0$	D_1	$-2+x_5\neq 0$
	D_1	$-1+x_1\neq 0$
$3+x_1+x_2+-3*x_4=0$	E_2	$x_1=-4-x_3+6*x_4-x_5$
		$x_2=1+x_3-3*x_4+x_5$
	D_2	$7+2*x_3-9*x_4+2*x_5\neq 0$
$-1+x_2+x_8=0$	E_1	$x_8=1-x_2$
$-2-x_1-x_2+2*x_4\geq 0$	E_2	$x_1=2-x_3-x_5-6*s_1$
		$x_2=-2+x_3+x_5+3*s_1$
	E_3	$x_4=1-s_1$
	D_2	$-2+2*x_3+2*x_5+9*s_1\neq 0$
$4+x_2-x_3+x_6\geq 0$	E_2	$x_1=4-x_3+x_6-3*s_1-s_2$
		$x_2=-4+x_3-x_6+s_2$
		$x_5=-2-x_6-3*s_1+s_2$
	D_2	$-6+2*x_3-2*x_6+3*s_1+2*s_2\neq 0$
$11-x_5-x_6>0$	S_1[a]	$s_3=13+3*s_1-s_2$
	D_3	$s_3\neq 0$
$x_5+x_6\neq 0$	D_4	$-2-3*s_1+s_2\neq 0$
$4-4*x_4+x_5+x_6\geq 0$	S_1[a]	$s_1=2-s_2+s_4$
		$s_3=19-4*s_2+3*s_4$
$-x_4+x_9=0$	E_1	$x_8=1-x_2$
		$x_9=x_4$
$3+x_1+x_2\neq 0$	D_3	$s_3\neq 0$
		$3-3s_1\neq 0$
$3-x_3-x_4\neq 0$	D_2	$-6+2*x_3-2*x_6+3*s_1+2*s_2\neq 0$
		$2-x_3+s_1\neq 0$
$4-x_1-x_3+x_6+x_7\geq 0$	E_4	$x_7=-6+2*s_2-3*s_4+s_5$
$2+1/3*x_1+1/3*x_2-x_4+x_7\neq 0$	D_1	$-2+x_5\neq 0$
		$-1+x_1\neq 0$
		$1+x_7\neq 0$
$5-4*x_4+x_5+x_6-x_{10}=0$	E_3	$x_4=1-s_1$
		$x_{10}=1+s_4$
$3+x_2-x_3+x_6-x_{10}\geq 0$	E_0	$x_4=1$
		$x_9=1$
	E_1	$x_8=1-x_2$
	E_2	$x_1=4-x_3+x_6-s_2$
		$x_2=-4+x_3-x_6+s_2$
		$x_5=-2-x_6+s_2$
	E_3	$x_{10}=1+s_4$
	E_4	$x_7=-2-s_4+s_5$
	S_0[a]	$s_1=0$
		$s_6=0$
	S_1[a]	$s_2=2+s_4$
		$s_3=11-s_4$
	D_1	$-2+x_5\neq 0$
		$-1+x_1\neq 0$
		$1+x_7\neq 0$
		$2-x_3\neq 0$
	D_2	$-6+2*x_3-2*x_6+2*s_2\neq 0$
	D_3	$s_3\neq 0$
	D_4	empty

[a] Result after using the simplex. s_4 is a new s-variable. Therefore, there is no need to dereference D_4.

< *dictionary* > ::= [< *var-list* > , < *status* > , < *eq-syst* > , < *s-syst* > , < *diseq-syst* >]
< *var-list* > ::= [ ] | < *var-element* > {, < *var-element* > }†
< *status* > ::= [< *rank-of-last-s-var* > , < *s-var-flag* >]
< *rank-of-last-s-var* > ::= *integer*
< *s-var-flag* > ::= *new*|*old*
< *var-element* > ::= (*alphanumeric*, < *variable-id* >)
< *eq-syst* > ::= [< E_0 > , < E_1 > , < E_2 > , < E_3 > , < E_4 >]
< *diseq-syst* > ::= [< D_0 > , < D_1 > , < D_2 > , < D_3 > , < D_4 >] } see Section 2.
< *s-syst* > ::= [< S_0 > , < S_1 >]
< E_0 > ,..., < E_4 > *are* < *lists-of-constraints* >
< D_0 > ,..., < D_4 > *are* < *lists-of-constraints* >
< S_0 > *and* < S_1 > *are* < *lists-of-constraints* >
< *list-of-constraints* > ::= [ ] | < *constraint* > {, < *constraint* > }

< *constraint* > ::= (< *pattern* > , < *left-hand-side* > , < *predicate* > , < *right-hand-side* >)
< *pattern* > ::= [< *constant-coef* > , < *n°-x-var* > , < *n°-s-var* > , < *n°-neg-s-var-coef* >]
< *left-hand-side* > ::= < *linear term* >
< *predicate* > ::= > | ≥ | ≠ | = | ≤ | <
< *right-hand-side* > ::= [ ] | < *linear-term* > {, < *linear-term* > }
< *constant-coef* > ::= *zero*|*non-zero*
< *n°-x-var* > ::= *integer*
< *n°-s-var* > ::= *integer*
< *n°-neg-s-var-coef* > ::= *integer*

< *linear-term* > ::= (< *type* > , < *variable-id* > , < *rational* >) | < *constant* >
< *type* > ::= *x-var*|*s-var*
< *variable-id* > ::= *integer*
< *constant* > ::= (*const*, < *rational* >)
< *rational* > ::= [< *numerator* > , < *denominator* >]
< *numerator* > ::= *integer*
< *denominator* > ::= *integer*

FIGURE 4. Internal data structure. The dagger indicates that the notation {*a*} corresponds to a (possibly empty) sequence of *a*s.

- The list < *status* > represents the pair [< *rank-of-last-s-var* > , < *s-var-flag* >], where the flag is used to bypass the dereferencing of D_4 when no fixed variables are detected after execution of the simplex. The pair < *status* > is used in implementing the optimization described in the previous section.
- The classification of solved forms can be done by examining the following data:
  1. The number of x-variables in the r.h.s., e.g., 0 for E_0, 1 for E_1, greater than 1 for E_2, etc.
  2. The number of s-variables in the r.h.s., e.g., to distinguish between different E and S classes.

For the purpose of quickly testing the unsatisfiability of the S class of constraints the preceding information is stored using a < *pattern* > .

Figure 6 describes the top-level procedures for solving constraints that will be called by the metalevel interpreter (see Section 4). We only describe the top-level procedures called by `solve_num_constraint`. The informal description pro-

Assumed first constraint: $2/3 + x_1 + 2 * x_2 - x_3 = x_4$
representation in the dictionary:
< *var-list* > : [(x_1, 1), (x_2, 2), (x_3, 3), (x_4, 4)]
< *rank-of-last-s-var* > : 0
< *s-var-flag* > : old
< *pattern* > : [1, 3, 0, 0]
constraint in E_2:
$x_1 = -2/3 - 2 * x_2 + x_3 + x_4$
(x-var, 1, [1, 1]), =, [(const, [−2, 3]), (x-var, 2, [−2, 1]),
(x-var, 3, [1, 1]), (x-var, 4, [1, 1])]

Assume second constraint: $4/3 * x_6 \geq x_2 - x_3$
representation in the dictionary:
< *var-list* > : [(x_1, 1), (x_2, 2), (x_3, 3), (x_4, 4), (x_6, 5)]
< *rank-of-last-s-var* > : 1
< *s-var-flag* > : new
< *pattern* > : [0, 2, 1, 1]
constraint in E_2:
$x_2 = x_3 + 4/3 * x_6 - s_1$
(x-var, 2, [1, 1]), =, [(x-var, 3, [1, 1]), (x-var, 5, [4, 3]),
(s-var, 1, [−1, 1])])

Assumed third constraint: $x_3 - x_6 \neq 3/4$
representation in the dictionary:
< *var-list* > : [(x_1, 1), (x_2, 2), (x_3, 3), (x_4, 4), (x_6, 5)]
< *rank-of-last-s-var* > : 1
< *s-var-flag* > : new
< *pattern* > : [1, 2, 0, 0]
constraint in D_2:
$0 \neq 3/4 - x_3 + 1/2 * x_6$
(const, [0, 1]), ≠, [(const, [3, 4]), (x-var, 1, [−1, 1]),
(x-var, 2, [1, 2])])

FIGURE 5. Examples of internal data structures.

vided in Figure 6 should enable the reader to reconstruct those procedures, if so desired.

4. INCORPORATION TO A METALEVEL INTERPRETER

In the previous sections we presented Prolog procedures to incrementally test the satisfiability of systems of linear constraints. The backtracking capabilities of Prolog can therefore be advantageously used in incorporating the test of satisfiability to a metalevel interpreter.

The nucleus of the interpreter is based on the approach described in [10]. The AND-OR graph corresponding to the nucleus is presented in Figure 7; the nucleus itself appears in Figure 8. The nucleus in Figure 7 can handle trees whose leaves may be atoms, variables, or linear constraints.

The main procedure `solve` calls `solve_goal` to handle individual goals or constraints, and recursively calls itself to process the remaining goals. Goals or constraints are processed by first checking their type. If the constraints are numeric, they are handled by `solve_num_constraint`. Tree constraints are processed by `solve_tree_constraint`. Finally, user-defined predicates are

```
% solve_linear_constraint(+CONSTR, +PREV_DICT, -UPD_DICT)
% This procedure embodies Steps 1, 2, and 3 in Figure 2.
solve_linear_constraint(Constr, Dict, New_Dict):-
   Dict=(Var_List, State, Eq_Syst, S_Syst, Diseq_Syst),
   internal_rep(Constr, Var_List, Coded_Constr, New_Var_List),
   Temp_Dict=(New_Var_List, State, Eq_Syst, S_Syst, Diseq_Syst),
   deref(Coded_Constr, Temp_Dict, Deref_Constr),
   test_satisfiability(Deref_Constr, Temp_Dict, New_Dict).

% internal_rep (+EXTERNAL_CONSTR, +VAR_LIST, -CODED_CONSTR, -UPD_VAR_LIST)
```

An external constraint is transformed into its internal representation using the variable list. If necessary, the variable list is then updated.

```
% deref(+CODED_CONSTR, +DICT, -DEREF_CONSTR)
% This procedure embodies the optimized step 2 shown in Figure 3 and it
also performs simplifications.
```

The procedure `deref` *utilizes* `Dict` *to dereference the coded constraint, transforming it into a dereferenced constraint. Notice that the* < *pattern* > *of the constraint may be changed after dereferencing (see Figure* 1).

```
% test_satisfiability(+CONSTR, +PREV_DICT, -UPD_DICT)
% This procedure embodies Step 4 in Figure 2.
```

It classifies a constraint into one of the three classes:

(1) *disequality involving x and s-variables.*
(2) *equality involving x-variables.*
(3) *equality involving only s-variables.*

In cases (1) *and* (2) *the constraint is incorporated to the appropriate class, if it does not imply in an inconsistency between* E_0 *and* D_1. *In case* (3) *the modified simplex is invoked.*

FIGURE 6. Top-level procedures.

handled by `solve_user_pred`. It is important to note that the latter may generate additional constraints corresponding to the unification of goal with the head of an applicable rule. Tree constraints are handled in a classical manner. First, variables in a tree constraint are dereferenced using the definition stored in a dictionary of tree constraints. Once dereferenced, the constraint is either incorporated to the dictionary (if it defines a new variable) or it is recursively processed by generating new constraints followed by a call to `solve`.

We have adopted the following conventions for input programs. Rational (numeric) constraints are specified by the term of the form

`nc(` < *list of constraints* > `).`

For example, $x + y > 8$ and $y \neq 3$ are expressed by

```
nc(x+y>8, y<>3).
```

This is necessary to avoid ambiguous situations. Note, however, that numeric constraints can also be generated by matching predicates and using `num(`V`)` to denote that V is either a numeric variable, a constant, or a numerical term. For example, the matching of

```
p(num(2), num(x+y))   and   p(num(a), num(3))
```

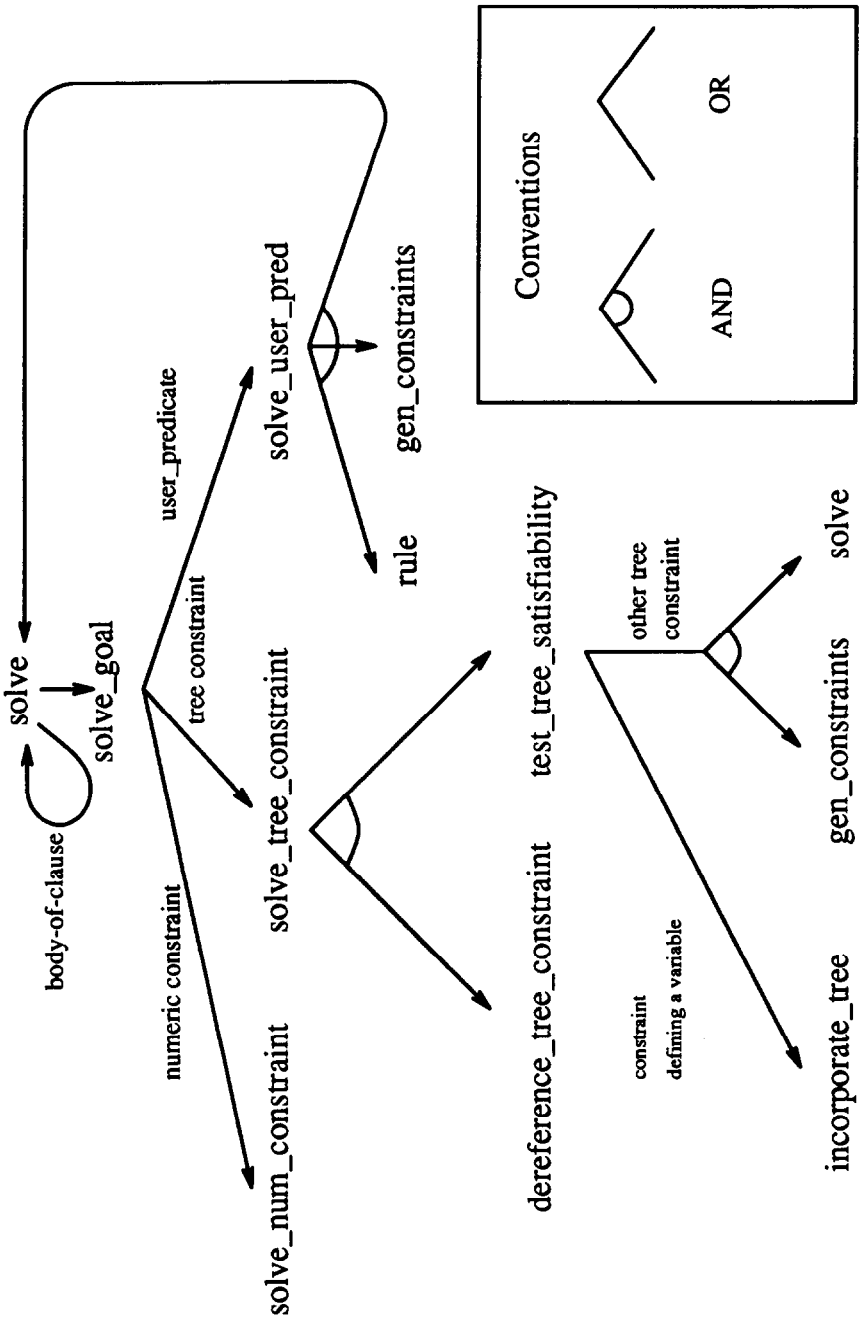

FIGURE 7. AND-OR graph of the main calls of the metalevel interpreter.

```
% solve(+GOAL_LIST, +PREV_DICT, -UPD_DICT)
solve([ ], Dict, Dict):-print(Dict).
solve([Goal|Rest], Prev_Dict, Upd_Dict):-
   solve_goal(Goal, Prev_Dict, Temp_Dict),
   solve(Rest, Temp_Dict, Upd_Dict).

% solve_goal(+GOAL, +PREV_DICT, -UPD_DICT)
solve_goal(Num_Constr, Prev_Dict, Upd_Dict):-
   solve_num_constraint(Num_Constr, Prev_Dict, Upd_Dict).
solve_goal(Tree_Constr, Prev_Dict, Upd_Dict):-
   solve_tree_constraint(Tree_Constr, Prev_Dict, Upd_Dict).
solve_goal(Goal, Prev_Dict, Upd_Dict):-
   solve_user_pred(Goal, Prev_Dict, Upd_Dict).

% solve_num_constraint(+NUM_CONSTR, +PREV_DICT, -UPD_DICT)
% merge with previous numeric constraints.
solve_num_constraint(Num_Constr, [Tree_Dict, Num_Syst],
  [Tree_Dict, New_Num_Syst]):-
   Num_Constr=..[nc| List_of_Constraints],
   !,
   solve_num_syst(List_of_Constraints, Num_Syst, New_Num_Syst).

% solve_tree_constraint(+CONSTRAINT, +PREV_DICT, -UPD_DICT)
% Merge constraints
solve_tree_constraint(Constr, Prev_Dict, Upd_Dict):-
   dereference_tree_constraint(Constr, Temp_Dict, NewConstr),
   test_tree_satisfiability(NewConstr, Temp_Dict, Upd_Dict).

% solve_user_pred(+GOAL, +PREV_DICT, -UPD_DICT)
% Determine applicable rule
solve_user_pred(Goal, Prev_Dict, Upd_Dict):-
   rule(Head, Body),
   gen_constraints(=, Head, Goal, Eqs),
   solve(Eqs, Prev_Dict, Temp_Dict),
   solve(Body, Temp_Dict, Upd_Dict).
```

FIGURE 8. Nucleus of the metalevel interpreter.

automatically generates

```
nc(2=a, x+y=3).
```

5. EXAMPLES

In this section we present two representative examples that have been frequently utilized in the description of Prolog III [7]. The first is a program computing the list of installments needed to repay a borrowed amount C with a given interest rate that in this particular example is 10%, but it could easily be generalized to any rate.[6]

[6]The program has been edited to improve readability. In Section 4, we described the terms `nc( )` and `num( )` that are actually listed by the interpreter. Notice that the tree representation for $[H|T]$ is `list(`H`,`T`)`.

5.1. *Mortgage*

```
% mortgage(INSTALLMENT, AMOUNT)
mortgage([ ], 0).
mortgage([A|Rest], C):-mortgage (Rest, (1+10/100)*C-A).
```

The program is presented in the interpreter in the form of unit clauses `rule(Head, Body)`. Consider the query:

```
? mortgage ([v, 2*v, 3*v], 1000).
```

The tree constraints that are incorporated to the tree dictionary and the numeric constraints processed by the numerical solver are summarized in the Table 2. The results are obtained by printing the entire dictionary of constraints. The presentation can be improved by developing a more sophisticated printing program having access to the list of variables appearing in the query. In that case, the final result becomes

```
v=133100/641.
```

5.2. *Periodicity*

The purpose of the second example is to show that the sequence of numbers

$$x_i, x_{i+1}, x_{i+2}, \ldots,$$

such that

$$x_{i+2} = |x_{i+1}| - x_i,$$

has a period 9.

The program establishes linear constraints corresponding to the specification of absolute values:

```
abs_value(X, X):-X>=0.
abs_value(X, -X):-X<0.
```

TABLE 2. Constraints incorporated to the tree dictionary or processed by the numeric solver.

Tree Constraints	Linear Constraints
	$a_1 = v$
$R_1 = [2*v, 3*v]$	
	$c_1 = 1000$
	$a_2 = 2*v$
$R_2 = [3*v]$	
	$c_2 = (1 + 1/100)*c_1 - a_1$
	$a_3 = 3*v$
$R_3 = [\ ]$	
	$c_3 = (1 + 1/100)*c_2 - a_2$
	$0 = (1 + 1/100)*c_3 - a_3$

To show that the sequence has period 9, the program considers the sequence

$x_1, x_2, x_3, \ldots, x_9, x_{10}$

and generates the constraints

$x_{i+2} = |x_{i+1}| - x_i \quad \text{for} \quad 1 \leq i < 9,$

$x_1 \neq x_{10}.$

The first nine constraints are generated by the program

```
sequence([U, V]):-U< >V.
sequence([U, V, Z|Rest]):-U=S-Z,
                          sequence([V, Z|Rest]),
                          abs_value(V, S).
```

The query

```
?-sequence([X1, X2, X3, X4, X5, X6, X7, X8, X9, X10]),
   X1< >X10.
```

fails because of inconsistencies. In this second example, the detection of fixed variables is essential for the correctness of the results.

6. FINAL REMARKS

The essence of our proposed satisfiability test can be summarized as follows.

1. Inequalities are transformed into equalities and disequalities by considering two types of variables: those that are strictly positive (s-variables) and those that can be either positive or negative (x-variables).
2. A set of satisfiable constraints is subdivided into three classes:
   a. Disequalities expressing linear combinations of x- or s-variables.
   b. Equalities defining x-variables as functions of x- or s-variables.
   c. Equalities defining s variables in terms of s-variables.
3. The advantages of establishing the subclasses in 2 are:
   a. Dereferencing of a new constraint is performed incrementally using only the appropriate subset of the classes. This increases the algorithm's efficiency.
   b. Unsatisfiability is also detected incrementally by considering only the appropriate subclasses of equalities and disequalities. This also increases the algorithm's efficiency.

The foregoing strategy may well be applicable in testing the satisfiability of constraints in other domains. It is usually desirable to include both predicates of equality and disequality of elements of a domain. It is also the case that satisfiability of equalities is expressible in a solved form such as

variable $= f($*other variables*$)$,

meaning that the variable in the left-hand-side is defined in term of other variables that can themselves be assigned any values in the domain being considered. Similarly, disequalities are expressible by the function

$g($*list of variables*$) \neq \Phi.$

The general problem of efficiently and incrementally testing the satisfiability of constraints in a domain can be expressed as follows:

1. Determine disjoint classes of functions $f_1, f_2, \ldots, f_n$, defining the various types of variables. Also determine the corresponding subclasses $g_1, g_2, \ldots, g_n$ of disequalities.
2. Given a new constraint N dereference its variables using the variables defined in the appropriate subclasses.
3. Test the satisfiability of the dereferenced N using only the appropriate subclasses of equalities and disequalities. If unsatisfiability is detected, the algorithm fails. Otherwise, if N is an equality constraint, then one of its variables is chosen and the equality defining that variable is incorporated to one of the subclasses, unless that definition enters in conflict with a disequality subclass, in which case the algorithm fails. If N is a disequality, it is added to the proper subclass.

The nontrivial part of the preceding scheme consists of selecting the subclasses so as to use the minimum number of dereferences and inconsistency tests. Finally, we mention some of the possible avenues of research stemming from this work.

- Explore the use of partial evaluation in compiling CLP programs instead of having them interpreted.
- Investigate the use of abstract interpretation [3] in determining, at compile time, the specific subclasses needed to dereference a given constraint or to minimize the checks for unsatisfiability.
- Estimate possible speed-ups gained by an OR-parallel execution of the metalevel interpreter.
- Extend the present interpreter to consider nonlinear constraints that are processed (using lazy evaluation) with the expectation that they become linear [8]. This is the approach used in both CLP ($\mathscr{R}$) and in Prolog III.

To summarize, the availability of an operational metalevel interpreter written in Prolog makes it possible to experiment with a variety of techniques that are presently available to increase the efficiency of Prolog programs but could be generalized to include CLP programs.

The authors wish to acknowledge the comments provided by Alain Colmerauer and Pascal Van Hentenryck.

REFERENCES

1. Abramson, H. and Rogers, M. H., *Meta-Programming in Logic Programming*, MIT Press, Cambridge, MA, 1989.
2. Bland, R. G., New Finite Pivoting Rules for the Simplex Method, *Math. Oper. Res.* 2:2 (1977).
3. Bruynooghe, M. et al., Abstract Interpretation Towards the Global Optimization of Prolog, *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, 1987.
4. Chvatal, V., *Linear Programming*, W. H. Freeman, San Francisco, CA, 1983.

5. Cohen, J., Constraint Logic Programming Languages, *Commun. ACM* 33:7 (1990).
6. Cohen, J., Koiran, P., and Perrin, C., Meta-Level Interpretation of CLP (Linear Lists, Integers) Enhanced by Lazy Evaluation and Enumeration, in: A. Colmerauer and F. Benhamou (eds.), *Constraint Languages*, MIT Press, Cambridge, MA, 1992.
7. Colmerauer, A., An Introduction to Prolog III, *Commun. ACM* 33:7 (1990).
8. Colmerauer, A., Résolution naive de contraintes non-linéaires, in: A. Colmerauer and F. Benhamou (eds.), *Constraint Languages*, MIT Press, Cambridge, MA, 1992.
9. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Betherier, F., The Constraint Logic Programming Language CHIP, *Proceedings of the International Conference on Fifth Generation Computing Systems*, 1988.
10. Hickey, T., Cohen, J., and Deschamps, V., Meta-Level Interpretation of Constraint Languages. A Case Study: Logical Programming, *New Generation Computing* 10:4 (1992).
11. Imbert, J. L. and Van Hentenryck, P., Efficient Handling of Discquations in CLP over Linear Rational Arithmetic, in: A. Colmerauer and F. Benhamou (eds.), *Constraint Languages*, MIT Press, Cambridge, MA, 1992.
12. Jaffar, J. and Michaylov, S., Methodology and Implementation of a CLP System, in: J. L. Lassez (ed.), *Proceeding of the 1987 International Logic Programming Conference*, MIT Press, Cambridge, MA, 1987.
13. Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R., The CLP ($\Re$) Language and System, IBM Research Report, T. J. Watson Research Center, 1990.
14. Komorowski, J., Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog, *Proceedings of the 9th Annual ACM Principles of Programming Languages Conference*, 1982, pp. 255–267.
15. Lassez, J. L. and McAloon, K., A Canonical Form for Generalizing Linear Constraints, *J. Symbolic Comput.* 13:1–24 (1992).
16. Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Yale University, June 1991, *Sigplan Notices* 26:9 (1991).
17. Schrijver, A., *Theory of Linear and Integer Programming*, Wiley, New York, 1987.
18. Van Hentenryck, P. and Graf, T., Standard Forms for Rational Linear Arithmetic in Constraint Logic Programming, presented at the International Symposium on Mathematics and Artificial Intelligence, Fort Lauderdale, Florida, January 1990.